The
BRITISH UNIVERSITY
IN EGYPT

CSCI05I

Logical and Artificial Intelligence

Dr. Amr Ghoneim

TA. Nouran Alaa

TA. Omar Khaled

Phase (2) Report

An **Intelligent Connect-Four Player** using the Minimax Algorithm, Alpha-Beta Pruning, and Heuristic Functions

Group 17

Names	ID
Mahmoud Mohamed	236360
Amr Mohamed Galal	235355
Yousef Moustafa	232157
Habiba Amr	235133

Project Overview:

This project overview involves the design of an intelligent Connect-Four player heavily reliant on the different key artificial intelligence techniques for implementation. Players alternately drop their colored discs into a grid, struggling to achieve four in a row before their opponent does. This project features a Minimax algorithm and Alpha-Beta pruning with well-crafted heuristic evaluation functions competing against human players under the computer's control. An amalgamation of different AI methodologies allows the system to efficiently decide on the most promising moves while adapting rather fluidly across different difficulty levels. Strategic decision-making highly relies on AI which project proponents purport to be highly valuable and will help users fine-tune their reasoning abilities in an almost real-life simulated environment.

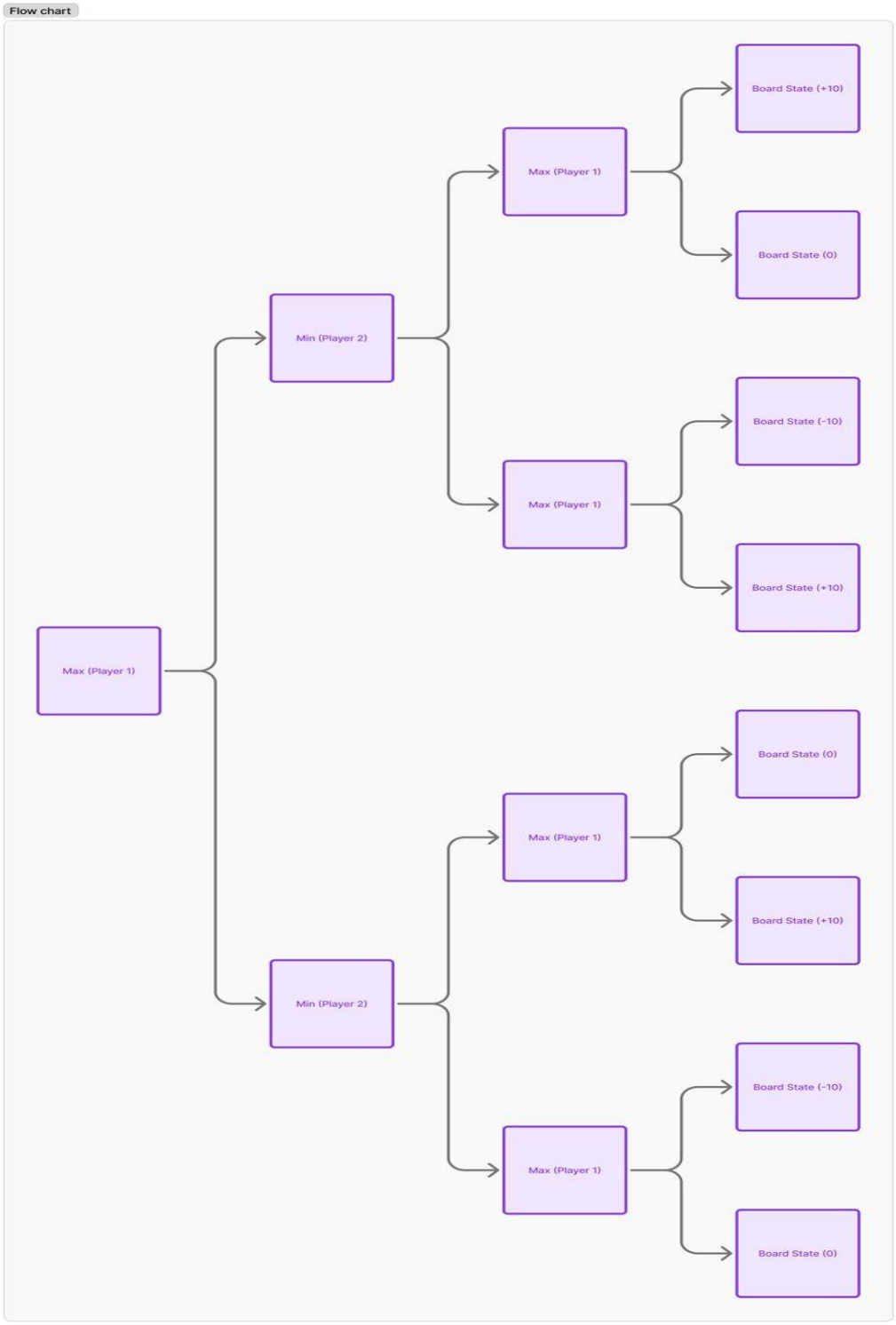
Main Functionalities;

A human player competes pretty vigorously against AI via simple interactive system interface remarkably well with some ease. The board dynamically updates after every move. The Connect-Four AI extensively makes use of the Minimax algorithm with Alpha-Beta pruning to simulate possible future moves and effectively select the best action. Two quite different heuristic functions were developed and then compared with each other yielding an implicit comparative analysis. AI examines non-terminal board states and decently decides what would be the most advantageous moves, without every time searching through the entire game tree in its entirety. The player can choose from various challenging modes, each with different difficulty levels. At higher difficulty levels, deeper search depths are made possible, and the AI appears to be more cunning, which somewhat leads to strategic gameplay. The systems illuminate AI moves during game progress, enabling players to follow the AI decision-making process quite closely now. Performance metrics such as win rates and average decision time have already been looked into for measuring AI effectiveness pretty thoroughly.

Methodology:

This is the algorithm of decision making for two-player games: the Minimax algorithm with alpha-beta pruning. It explores recursively all possible moves up to a certain depth or until the game ends, trying to maximize that of the AI while minimizing the opponent's chances. If the AI is the current player (max), it will choose the one with the highest score in the end; if it is the human player (min), she will select the one with the lowest score. An alpha value or beta defines the cut-and-prune approach because it excludes the parts of the search that would eventually not alter the decision making. It also speeds up the search by judging moves irrelevant to the outcome. The terminal states are win loss draw states and such board evaluations point the AI to the right move.

Diagram: Minimax Flowchart



Pseudo Code for MiniMax Algorithm:

```
if (depth == 0 or the board is a terminal node) {
    Evaluate and return the board score
}
else if (maximizingPlayer) {
    Initialize value = -∞
    For each valid move {
        Simulate dropping a piece
        Call Minimax(recursive) with depth-1, minimizingPlayer
        if (new score > value) {
            Update value and best column
        }
        Update alpha = max(alpha, value)
        if (alpha >= beta) {
            Break (prune branch)
        }
    }
    return best column and value
}
else {  // minimizingPlayer
    Initialize value = +∞
    For each valid move {
        Simulate dropping a piece
        Call Minimax(recursive) with depth-1, maximizingPlayer
        if (new score < value) {
            Update value and best column
        }
        Update beta = min(beta, value)
```

```
        if (alpha >= beta) {
            Break (prune branch)
        }
    }
    return best column and value
}
```

Justification:

We decided to organize the Connect-Four project using a method involving Python, NumPy, and Pygame, since each tool directly addresses the needs of an intelligent game agent with a graphical interface. NumPy was selected for representing the game board, as it facilitates efficient manipulation of two-dimensional arrays, thus simplifying operations like checking for valid moves, placing pieces, and detecting winning conditions. Pygame was used for the interface as it has powerful functions for real-time graphical rendering and event handling, allowing an interactive user experience without unnecessary complexities. The core game logic was built into fairly clear and modular functions, such as drop_piece, is_valid_location, and winning_move, to preserve the clean separation of different functionalities and let an understanding of code be easier to debug and extend. The Minimax algorithm with Alpha-Beta pruning is implemented for the AI agent so that it always makes moves optimally due to the reduced time in decision-making, thereby keeping the AI competitive without compromising its performance. The architecture was therefore considered to strike a balance between readability, efficacy, and AI intelligence for creating a Connect-Four player that is truly being a challenge and fun for human users.

Implementation Details:

def create_board()

This function creates the Connect-Four game board. It uses NumPy to create a 2D array (6 rows × 7 columns) filled with zeros, where zero represents an empty slot. This function is used to Initialize an empty board at the start of the game using NumPy zeros() function.

def drop_piece(board, row, col, piece):

This function places a piece on the board. It updates the board array by setting a specific location (row, col) to either PLAYER_PIECE (1) or AI_PIECE (2). This function is used to actually place the player's or AI's move on the board using simple Python array assignment.

def is_valid_location(board, col):

This function checks if a move is valid. It looks at the top row of the given column to see if it's empty (0). If it is empty, a piece can still be dropped in that column. This function is used to prevent players from dropping pieces into full columns using Python comparison (==) and array access

def get_next_open_row(board, col):

This function finds the next available (empty) row in a selected column. It starts from the bottom and moves upward until it finds the first empty cell. This function is used to make pieces "fall" to the lowest available position in a column using python for loop and condition checking

def print_board(board):

This function prints the current state of the board to the console. It uses NumPy's flip() function along the vertical axis (axis 0) to reverse the board — because in Connect-Four, pieces fall to the bottom, but arrays start at the top. This function is used to display the board correctly in a readable way using NumPy flip() and Python print() function.

def winning_move(board, piece):

This is a critical function that checks if the current board has a winning move for the specified piece (either player or AI), it detect if a player or AI won the game. It checks four types of winning possibilities:

Horizontal win: 4 connected pieces side by side across a row.

Vertical win: 4 connected pieces stacked in a column.

Positive slope diagonal win: 4 pieces connected diagonally upwards.

Negative slope diagonal win: 4 pieces connected diagonally downwards.

Each check uses a nested for loop and Python's all() function to see if four pieces in a line are all the same (piece).

def evaluate_window(window, piece):

This function evaluates a group of four cells ("window") on the game board and scores it based on how favorable it is for the player. It gives a high positive score if the player is close to winning, a small score for partial progress, and a negative score if the opponent is close to winning. This helps the AI or player prioritize good moves and block threats using Python list count() method.

def score_position(board, piece):

This function evaluates the entire game board and computes a score for a given player. It rewards occupying the center column and systematically examines every possible 4-cell window horizontally, vertically, and diagonally. Each window is scored by the evaluate_window function, and the scores are summed to guide the AI or player toward better positions. It uses NumPy slicing (board[:, c], board[r, :]) and list operations.

def get_valid_locations(board):

This function checks each column of the board and returns a list of columns where a move is possible (columns that are not yet full). It uses a list comprehension and the is_valid_location helper function to filter out invalid columns.

def is_terminal_node(board):

This function checks if the current board state is a terminal node which means the game is over. A board is terminal if either the player or AI has won, or if there are no more valid moves (draw). It combines win detection and move availability in a single check.

def minimax(board, depth, alpha, beta, maximizingPlayer):

```
def minimax(board, depth, alpha, beta, maximizingPlayer):
    valid_locations = get_valid_locations(board)
    is_terminal = is_terminal_node(board)
    if depth == 0 or is_terminal:
        if is_terminal:
            if winning_move(board, AI_PIECE):
                return (None, 100000000000)
            elif winning_move(board, PLAYER_PIECE):
                return (None, -100000000000)
            else:
                return (None, 0)
        else:
            return (None, score_position(board, AI_PIECE))

    if maximizingPlayer:
        value = -math.inf
        best_col = random.choice(valid_locations)
        for col in valid_locations:
            row = get_next_open_row(board, col)
            b_copy = board.copy()
            drop_piece(b_copy, row, col, AI_PIECE)
            new_score = minimax(b_copy, depth-1, alpha, beta, False)[1]
            if new_score > value:
                value = new_score
                best_col = col
            alpha = max(alpha, value)
            if alpha >= beta:
                break
        return best_col, value
```

```
else:
    value = math.inf
    best_col = random.choice(valid_locations)
    for col in valid_locations:
        row = get_next_open_row(board, col)
        b_copy = board.copy()
        drop_piece(b_copy, row, col, PLAYER_PIECE)
        new_score = minimax(b_copy, depth-1, alpha, beta, True)[1]
        if new_score < value:
            value = new_score
            best_col = col
        beta = min(beta, value)
        if alpha >= beta:
            break
    return best_col, value
```

This function implements the Minimax algorithm with alpha-beta pruning to choose the best move. It recursively simulates all possible future moves up to a certain depth. If it's the maximizing player's turn (AI), it tries to maximize the score; if it's the minimizing player's turn (human), it tries to minimize the score. Alpha and beta values help cut off unnecessary calculations, making the search faster.

def draw_board(board):

This function draws the entire game board using pygame which is the GUI for our code. It first draws a blue background with black empty circles to represent slots using pygame.draw.rect(). Then, it draws the player's and AI's pieces (red and yellow circles) using pygame.draw.circle() based on the current board state. And displays the state when every turn is played using pygame.display.update()

def main():

The main() function controls the overall flow of the game and sets up and runs the entire Connect-4 game loop. It handles events like mouse movement to show where the piece will drop, mouse clicks to place the piece using pygame, also switching turns between player and AI, checking for wins, and updating the display. The AI uses the minimax function to choose its moves intelligently. Also we used random to decide who starts the player or the AI randomly.

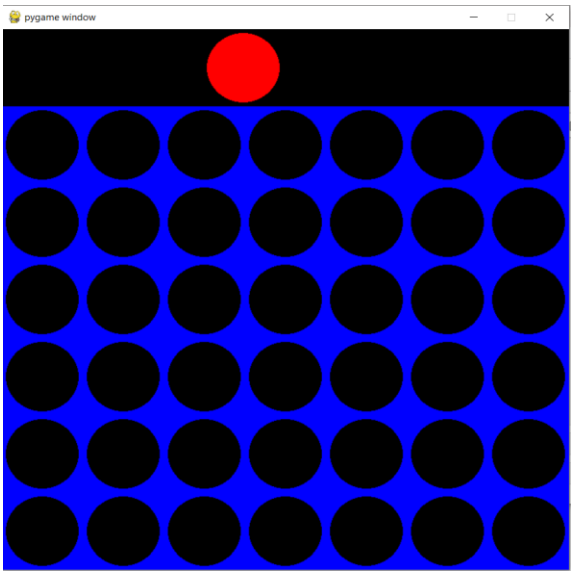
Testing and Evaluation:

This whole testing ended with the checking of individual components such that all functions like drop_piece, is_valid_location, and winning_move were all manually tested on small definitions of board configurations at initial stages, and this was done to make sure they were working well. Once that was completed, complete gameplay sessions were conducted in which a human was pitted against the artificial intelligence. There was some particular attention on the playing behavior of the AI to see whether it could identify winning moves, block its opponent from winning opportunities, and choose strategic moves, such as taking the middle column.

Screenshots were taken of the AI during the tests, showing when it successfully made a four-piece connection or if it prevented an opponent from winning. Metrics used to measure its performance were mainly the AI's win rate over several games, time taken per move, and the quality of decisions made at each search depth. This AI displayed a strong performance with lower depths and responsiveness with Alpha-Beta Pruning, even with slightly greater searching. A comparison of the various heuristic functions revealed differences in aggressiveness between getting a win or blocking an opponent. It was shown that the AI made intelligent and timely decisions, suggesting strategic understanding of Connect Four.

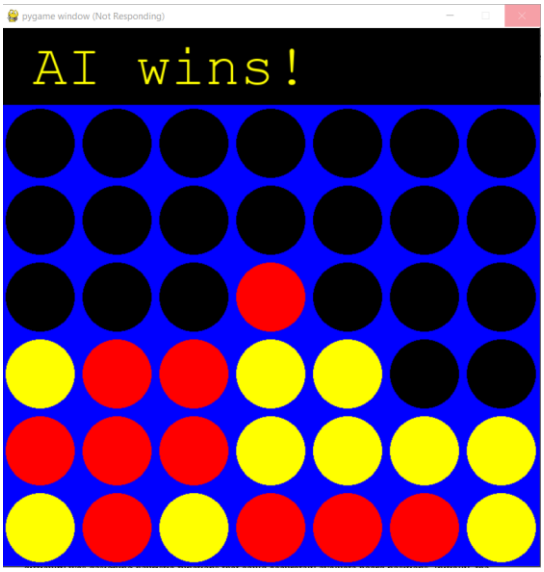
Initial Board State:

At first, the board is initially empty and the player starts to play and choose where to put the red ball.



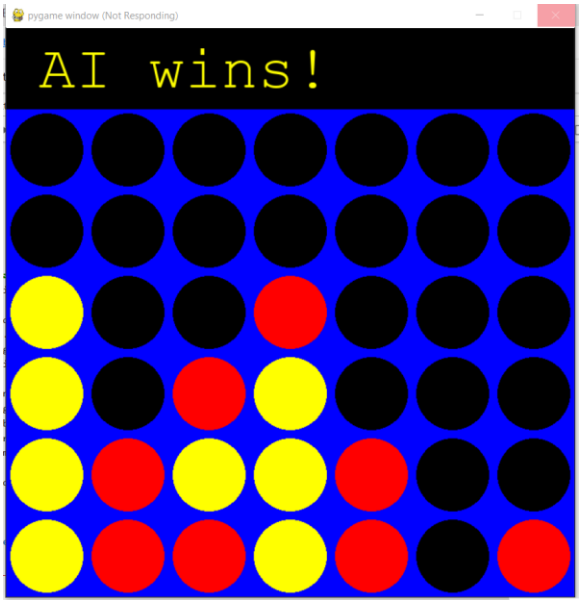
AI winning States:

1)



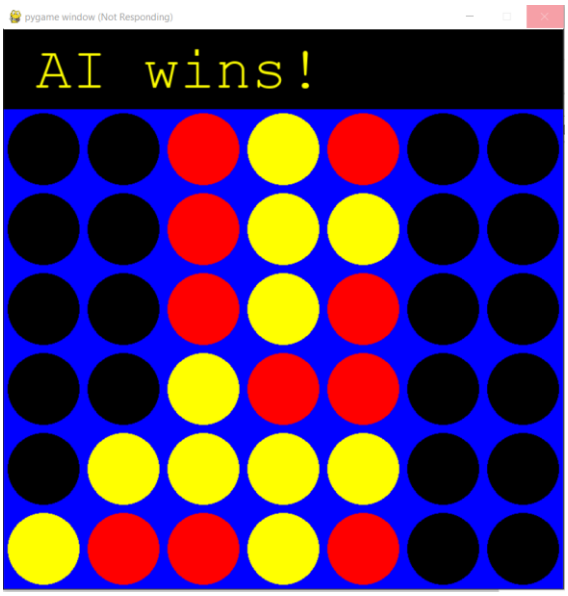
Here in this state the AI won, the user and AI played the game until AI reached 4 yellow balls beside each other in the same row, so AI won and a message pops up saying “AI wins!”

2)



Here AI won also, but with 4 yellow balls on each other vertically and a message pops up: “AI won !”

3)



Here AI won also, but with 4 yellow balls that are diagonally and a message pops up: “AI won !”

Challenges & Future Work

Somewhat frustratingly, balancing the depth of Minimax search with the performance of the program was a major challenge during the project development. Deep-level AI systems analyze much potential future moves at excessive computational costs such that the response time shows a significant delay. The implementation of Alpha-Beta Pruning has helped to an extent in reducing unnecessary node evaluations, yet, performance tuning at deep levels remains a more challenging task. Designing heuristic functions capable of accurately evaluating board positions formed another significant difficulty during the process somewhat frustratingly. The evaluation function was sending the AI down a garden path of suboptimal moves with frightening frequency under certain situations during the game. Recent tweaks received much focus on rewarding plays in the central column and punishing moves that easily allowed an opponent to win.

Future work may involve implementing such advanced heuristic evaluations that it could double AI's foresight without searching significantly deeper. By varying search depth according to selected difficulty, adjustable difficulty levels could be incorporated in a way that greatly affects the gameplay experience. Some enhancements may involve more engaging gameplay, with animations coming into play after each move and ridiculously better visual feedback. Strong foundations were thus laid for a smart Connect-Four player with much potential for further AI refitting and user experience tweaks.